Informatique Commune

Rappels du lycée et quelques compléments

Erik Jacob

10 septembre 2021

Table des matières

I Introduction

I.1 Notion d'algorithme

Définition 1. Un **algorithme** est une séquence finie et ordonnée d'opérations qui, prenant un ensemble (fini) de valeurs en entrée, produit un ensemble (fini) de valeurs en sortie.

▶ *On peut penser à une notice de montage de meuble, une méthode de calcul d'une fonction dérivée, etc.*

Définition 2. Un **langage de programmation** est un ensemble de notations destinées à formuler des algorithmes. Il est muni d'une orthographe, d'un vocabulaire et d'une grammaire.

▶ *Selon un décompte réalisé par Bill Kinnersley, on compte de l'ordre de 2500 langages de programmation créés depuis les années 1950 jusqu'à nos jours. On peut citer :*

ADA	Algol	Basic	C	C++	C#
Cobol	Forth	Fortran	Haskell	HTML	Java
JavaScript	Lisp	ML	OCaml	Pascal	Perl
Prolog	Python	Ruby	Rust	SQL	VBA

Les langages peuvent être classés par familles selon les concepts mis en oeuvre - on parle de paradigmes. On peut citer quelques paradigmes que l'on détaillera partiellement au paragraphe suivant lors de la présentation de Python :

Impératif	Orienté objets	Fonctionnel
Événementiel	Orienté requêtes	Logique

Définition 3. Un **programme** est la traduction d'un algorithme dans un langage de programmation. Le programme dispose de données en entrée. Le programme va les traiter et produire des données en sortie. Toute phrase compréhensible traitée par le programme sera appelée **une ligne de code** ou **un code**.

— *Les entrées* sont des valeurs données avant que le programme ne commence. Ces valeurs vérifient des propriétés appelées des *préconditions*.

— *Les sorties* sont des valeurs produites par le programme. Ces valeurs vérifient des propriétés appelées des *postconditions*.

▶ *Prenons l'exemple du programme suivant :*

```
1 def factorielle(n):
2     '''renvoie factorielle n pour tout n>=0'''
3     assert n>=0,"n doit être positif !" #ligne non exigible si l'on
     suppose que la fonction est appelée avec n>=0
4
5     f = 1
6     for i in range(1, n + 1): # ou range(2,n+1)
7         f = f * i
8     return f
```

Ce programme dispose d'une donnée en entrée appelée n. Une précondition est que n soit un entier naturel. Une postcondition est que la valeur renvoyée par le programme soit l'entier $n!$
Notez que tout programme avec des données en entrée dispose de préconditions, à savoir que les entrées sont du type prévu pour le programme.

I.2 Présentation de Python

Python est un langage de programmation créé par Guido van Rossum en 1989 :

- *multi-plateforme et portable* : Windows, Linux, OS X,...,
- *gratuit et évolutif* : de licence libre,
- *interprété* : traduit ligne à ligne en langage machine au moment de l'exécution, contrairement aux langages compilés qui traduisent l'ensemble des programmes en langage machine en une seule fois,
- *de haut niveau* : plus proche du langage algorithmique que du langage machine,
- *à typage dynamique* : le type n'est pas spécifié par le programmeur, mais déduit par l'interpréteur.

Python est un langage de programmation vérifiant les paradigmes suivants :

- *impératif* : il ordonne les instructions en séquences pour modifier l'état de variables, en utilisant notamment des boucles (`while`,`for`),
- *orienté-objet* : il est possible de définir des objets avec des fonctions associées, appelées des *méthodes* (ex. pour un tableau t donné, on dispose de la méthode t`.append(valeur)`),
- *fonctionnel* : les fonctions peuvent avoir tout type d'objets en paramètre (entier, tableau, fonction,...) et elles renvoient en retour tout type d'objets (entier, tableau, fonction,...).

Python est utilisé par Google (Youtube est écrit en Python), Industrial Light & Magic (Lucas Film), la NASA, etc. Une des clés du succès de Python est d'être un langage assez dépouillé (un code Python est 3 à 5 fois plus court qu'un code correspondant écrit en C, C++ ou Java), entre-autre grâce à son typage dynamique et l'écriture indentée. Un de ses défauts, et c'est le revers du typage dynamique, est d'être gourmant en mémoire.

I.3 Framework

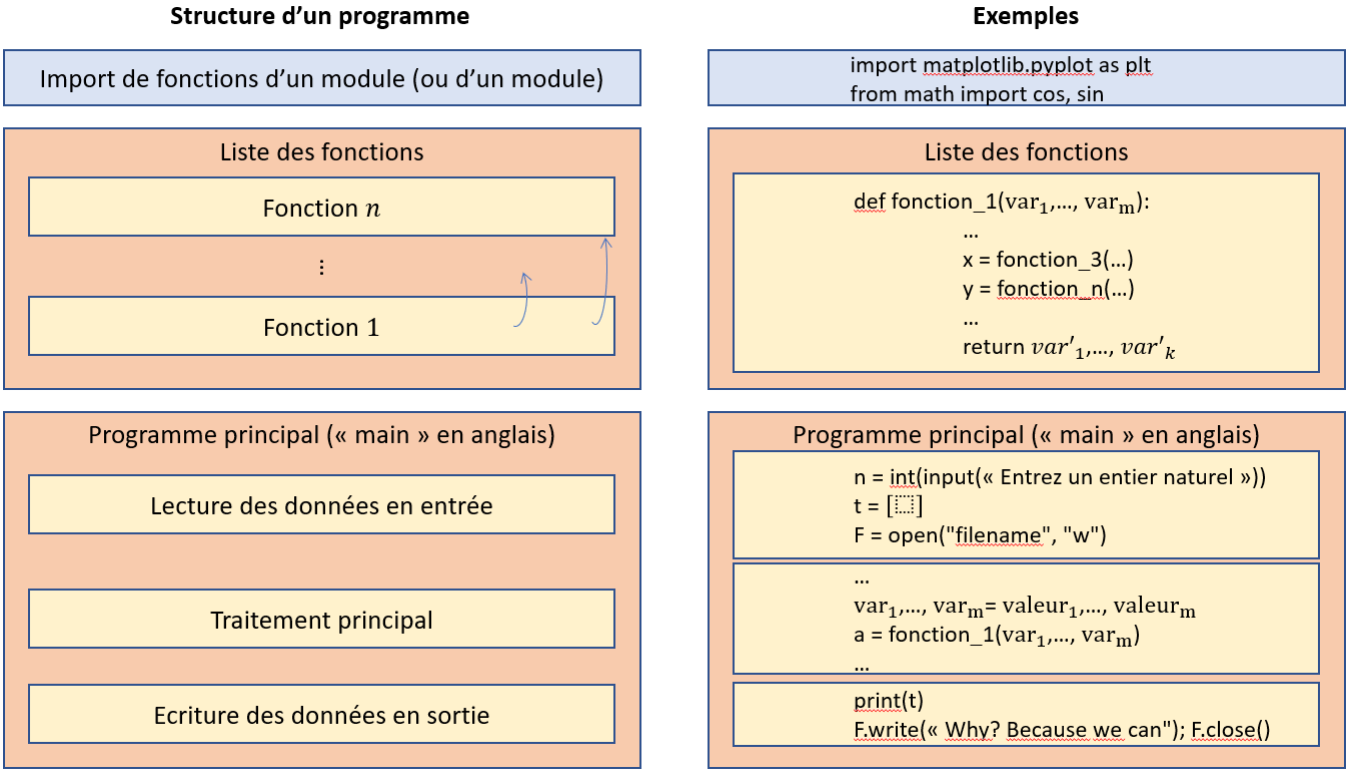
Le framework choisi au lycée Michelet pour travailler en Python est *Pyzo*. Il contient un environnement de développement intégré (IDE) appelé IEP (Interactive Editor for Python) ainsi qu'une version 3.* du langage Python ainsi qu'une série de modules. Il est nécessaire de distinguer deux composantes fondamentales de cet environnement : le *shell*, encore appelé "console" ou "terminal" (plusieurs peuvent tourner simultanément) et l'*éditeur de programme*.

Nous vous conseillons d'installer Python sur votre ordinateur personnel et d'utiliser une clé USB pour sauvegarder vos travaux personnels. Le plus simple est alors d'installer la même version que celle sur laquelle vous travaillerez au lycée : allez pour cela à l'adresse `http://www.pyzo.org`, et télécharger la version compatible avec le système d'exploitation que vous utilisez (Linux, Windows ou OS X).

I.4 La structure de base d'un programme en Python

La construction d'un programme suit une logique de **modularité** : cela consiste à découper son programme en fonctions, répondant à des objectifs précis. Chaque fonction doit *encapsuler* son code interne et ne dialoguer avec les autres fonctions qu'avec une interface : *en entrée*, une liste de paramètres (appelés aussi des **variables**) sur lesquels reposent éventuellement des contraintes précisés dans la documentation de la fonction, *en sortie* un ou plusieurs résultats disponibles par les autres fonctions.

Python est souple quant à l'organisation générale d'un programme, ce qui n'est pas le cas d'autres langages. Une bonne pratique est de structurer les programmes écrits en Python de la façon suivante :



Il faut savoir que l'interpréteur "lit" le programme de haut en bas pour l'interpréter.
Par contre, il l'exécute à partir du *main* (i.e. du traitement principal), en appelant les fonctions au fur et à mesure, tel qu'indiquées dans le code.

II Les types usuels

Lors de l'écriture de programmes, les informations utilisées sont mémorisées dans des variables. Pour pouvoir distinguer les différents contenus possibles et leur mode d'utilisation et de stockage, il est obligatoire de leur attribuer un **type**. Un certain nombre de types sont prédéfinis dans Python et nous allons présenter ceux déjà rencontrés dans les études secondaires.

II.1 Les types numériques : int, float, complex

Il y a trois types numériques :

- Le type `int` (abréviation d'*integer*) représentant les nombres entiers relatifs. En Python, le nombre de chiffres des entiers est **illimité** [1].
- Le type `float` représentant les nombres décimaux dits *à virgule flottante*, généralement stockés en double précision (64 bits).
- Le type `complex` représentant les nombres complexes dont les parties réelles et imaginaires sont des nombres flottants.

1. càd. uniquement limité par les capacités mémoire de la machine

Le tableau ci-dessous présente les opérations usuelles sur les types numériques (dans l'ordre des priorités croissantes).

`x+y`	somme de x et y
`x-y`	différence de x par y
`x*y`	produit de x par y
`x/y`	quotient *flottant* de x par y
`x//y`	quotient de la division euclidienne de x par y (pour y de type `int`)
`x%y`	reste de la division euclidienne de x par y (pour y de type `int`)
`-x`	opposé de x
`abs(x)`	valeur absolue/module de x
`z.conjugate()`	conjugué du nombre z
`x**y`	exponentiation de x par y

▶ *Exemples :*

```
>>> 2*5//6-4**4        #noter la priorité des opérations
-255

>>> 6/7                #division dans les flottants
0.8571428571428571

>>> 47/7;47%7          #quotient;reste de la division euclidienne
6.714285714285714
5

>>> abs((-2)**7)       #valeur absolue
128

>>> (1j)**2            #la notation j désigne le i mathématique
(-1+0j)

>>> (1+2j).real        #partie réelle
1.0

>>> (1+2j).imag        #partie imaginaire
2.0
```

▶ *Noter le typage dynamique :*

```
>>> x = 2             #x est automatiquement de type entier
>>> type(x)
<class 'int'>

>>> y = 3.4           #y est automatiquement de type float
>>> type(y)
<class 'float'>

>>> y = x            #y recupère la valeur d'une variable de type entier,
    donc devient dynamiquement un entier
>>> type(y)
<class 'int'>

>>> z = x/2  #la division entre entiers donne toujours un float
>>> type(z)
<class 'float'>

#   NB: si on souhaite continuer à travailler avec des entiers suite à
   une division, on écrit z = x//2 (partie entière) (ne pas écrire int(z
   /2))
```

II.2 Le type str

Le type `str` (abréviation de *string*) correspond aux *chaînes de caractères*. Une chaîne de caractères se définit de deux façons :

- **par extension** en encadrant la chaîne de caractères par le symbole " ou '
- **par conversion** à l'aide de la fonction de conversion `str`.

▶ *Exemples :*

```
>>> texte = "Hello world !"   #définition par extension avec ""
>>> type(texte)
<class 'str'>

>>> lettre = 'a'        #définition par extension avec ''
>>> c = ""              #définition par extension avec la chaîne vide

>>> str(12345)          #définition par conversion de l'entier 12345
'12345'
```

Les opérations usuelles sur les chaînes de caractères sont :

- le calcul de la longueur d'une chaîne de caractères,
- l'accès à un caractère d'une chaîne de caractères via son indice,
- l'extraction d'une sous-chaîne de caractères,
- la concaténation de deux chaînes de caractères,
- le test d'appartenance d'un caractère à une chaîne de caractères.

▶ *Le calcul de la longueur d'une chaîne de caractères :*

```
>>> texte = "Hello world !"
>>> len(texte) #len est l'abréviation de length qui signifie longueur
13
```

▶ *L'accès à un caractère d'une chaîne de caractères via son indice :*
Attention, en Python, **les indices démarrent par défaut à 0** !

```
>>> texte[0]; texte[12] #accès par indices positifs
'H'
'!'

>>> texte[-1]; texte[-13] #accès par indices négatifs
'!'
'H'
```

La structure d'une chaîne de caractères est la suivante :
Nommons n la longueur de la chaîne de caractères.

- Le premier caractère a pour indice 0 ou $-n$.
- Le dernier caractère a pour indice $n-1$ **(et non** n**)** ou -1.
- Le caractère d'indice i est le $(i+1)^{\text{e}}$ élément.

Indices	*0*	*1*	*2*	*3*				*n-4*	*n-3*	*n-2*	*n-1*
String	**H**	**e**	**l**	**l**		**...**		**l**	**d**		**!**
Indices	*-n*	*-n+1*	*-n+2*	*-n+3*				*-4*	*-3*	*-2*	*-1*

- *L'extraction d'une sous-chaîne de caractères :*
  On peut extraire une sous-chaîne de caractères par la technique du **slicing**.
  Le slicing **duplique** une tranche de la chaîne initiale :
  `texte[a:b]` crée la chaîne de caractères constituée des caractères `texte[a]` à `texte[b-1]` (donc ***b* est exclu**).

```
>>> texte = "Hello world !"
>>> nouveau_texte = texte[2:7] #slicing du texte
>>> nouveau_texte
'llo w'

>>> nouveau_texte = texte[1:8:2]  #le slicing fournit une copie de la variable texte du caractère 1 au caractère 7 par pas de 2
'el o'

>>>print(texte[2:])  #tranche de 2 à la fin de la chaine
'llo world !'

>>> print(texte[:7]) #tranche du début au 6e caractère inclus
'Hello w'

>>> print(texte[::-1]) #le slicing fournit une copie inversée de la variable texte et l'affiche
'! dlrow olleH'
```

- *La concaténation de deux chaînes de caractères :*
  On utilise tout simplement l'opérateur binaire + pour concaténer deux chaînes de caractères, càd. les mettre bout à bout.

```
>>> texte1 = "Bonjour "
>>> texte2 = "Arthur !"
>>> texte1 + texte2  #concaténation de texte1 et texte2
'Bonjour Arthur !'

>>> texte1 * 3   #triplication de texte1 (généralisation de la concaténation)
'Bonjour Bonjour Bonjour'
```

- *Le test d'appartenance d'un caractère à une chaîne de caractères :*
  On utilise le mot clé `in`.

```
>>> texte = "Bonjour"
>>> 'n' in texte  #le caractère 'n' est bien dans "Bonjour"
True

>>> 'N' in texte  #le caractère 'N' n'est pas dans "Bonjour"
False

>>> n in texte    #la valeur de la variable n n'est pas dans "Bonjour"...car cette variable n'est pas définie
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'n' is not defined
```

Contrairement aux tableaux que nous verrons ci-après, il n'est **pas** possible de modifier ou supprimer un caractère d'une chaîne de caractères (elle est dite *non mutable*).

```
>>> chaine = "amis"
>>> chaine[1] = 'v'
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

II.3 Le type list

Le type `list` correspond à des tableaux contenant une suite **finie** d'éléments **de type quelconque**, chacun d'eux étant référencé par un indice. En Python, les indices peuvent être positifs ou négatifs, selon un fonctionnement identique à celui des chaînes de caractères. En Python, on peut changer la valeur d'un élément d'un tableau, supprimer un élément d'un tableau (on les dit *mutables*) et accroître sa longueur à droite (c'est-à-dire du côté de l'indice le plus élevé).

Un tableau de type `list` peut être défini de plusieurs manières :

- **par extension** en encadrant le tableau par des crochets,
- **par compréhension** à l'aide de syntaxes intuitives,
- **par conversion** à l'aide de la fonction de conversion *list*.

▶ *Exemples :*

```
#Définition d'un tableau par extension
>>> tableau = [] #tableau vide
>>> tableau = [1,2,'3',[4,5]]
>>> type(tableau)
<class 'list'>

#Définition d'un tableau par compréhension
#première possibilité
>>> n = 15
>>> [0] * n
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

#deuxième possibilité
>>> n = 15
>>> [k for k in range(n)] #entiers successifs de 0 à n=15 exclu
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

>>> [k**2 for k in range(n)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]

#Définition d'un tableau par constructeur
list(range(n)) #range est un objet qui génère successivement les entiers
    de 0 à n=15 exclu
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Toutes les opérations usuelles vues au paragraphe précédent sur les chaînes de caractères s'appliquent également, **avec la même syntaxe**, sur les tableaux de type `list`. Cependant, on peut modifier la valeur d'un élément d'un tableau, ce qui n'est pas possible avec une chaîne de caractères.

▶ *Exemples :*

```
#Modification d'un tableau donné
>>> tableau = [1,2,3,4,5]

>>> tableau[0] = 2 #on rappelle que les éléments sont indicés à partir
   de 0
>>> tableau
[2, 2, 3, 4, 5]

>>> n = len(tableau)
>>> tableau[n-1] = 7 #le dernier élément est d'indice len(tableau)-1
>>> tableau
[2, 2, 3, 4, 7]
```

```
12
13 #Ajout d'un élément dans un tableau donné
14 #Première possibilité à privilégier
15 >>> tableau.append(8) #on ajoute 8 au tableau
16 >>> tableau
17 [2, 2, 3, 4, 7, 8]
18
19 >>> tableau.extend([9,10]) #on étend le tableau avec les éléments du
      tableau [9,10]
20 >>> tableau
21 [2, 2, 3, 4, 7, 8, 9, 10]
22
23 #Seconde possibilité
24 >>> tableau += [11] #on ajoute 11 au tableau donné
25 >>> tableau
26 [2, 2, 3, 4, 7, 8, 9, 10, 11]
27
28 #Copie d'un tableau donné avec ajout d'un nouvel élément
29 >>> tableau = tableau + [12] #la variable "tableau" est une copie du
      tableau donné, appelé "tableau", avec l'élément 12 en plus
30 >>> tableau
31 [2, 2, 3, 4, 7, 8, 9, 10, 11, 12]
32
33 #Slicing : création d'un tableau extrait d'un tableau donné
34 >>> tableau[2:5] #nouveau tableau créé à partir de "tableau"
35 [3, 4, 7]
36
37 >>> tableau #le tableau initial est inchangé
38 [2, 2, 3, 4, 7, 8, 9, 10, 11, 12]
39
40 #Suppression du dernier élément d'un tableau donné
41 >>> tableau.pop() # le dernier élément est supprimé et affiché
42 12
43 >>> tableau
44 [2, 2, 3, 4, 7, 8, 9, 10, 11]
```

Attention !
`append`, `extend` et `pop` sont appelées des **méthodes** : ce sont des fonctions canoniques associées au type `list`.
Quel que soit le nombre d'éléments du tableau, l'ajout des éléments se déroule en **temps constant** [2]. L'instruction `tableau+=[...]` se déroule également en temps constant, alors que ce n'est pas le cas de `tableau=tableau+[...]` qui lui est syntaxiquement très proche : il faudra ainsi veiller à limiter son utilisation (petits tableaux, ajout en tête lors d'une récurrence, ...).

II.4 Le type bool

Le type `bool` (du mathématicien George Boole) correspond aux deux valeurs logiques `True` (vrai) et `False` (faux). Les opérations logiques possibles sur une variable de type `bool` sont (par ordre de priorité croissante) la *disjonction* `or`, la *conjonction* `and` et la *négation* `not`.

▶ *Exemples :*

```
1 >>> type(0<1) #le résultat de l'inégalité est un booléen
2 <class 'bool'>
3 >>> 0<1
4 True
5 ##_____
```

2. Le temps est "à peu près" constant : ce point sera précisé ultérieurement en cours d'année.

```
>>> True==1 #le type bool est construit comme sous-type de int
True
>>> False==0 #True est 1 et False est 0
True
##------------------------------------------------------------------------
>>> not(True)
False
##------------------------------------------------------------------------
>>> 2==2.0 #le type int est converti en float
True
##------------------------------------------------------------------------
>>> (1<0) or (2<=3) #le second terme est vrai, donc le résultat est vrai
True
>>> (2<3) and (3<0) #le second terme est faux, donc le résultat est faux
False
##------------------------------------------------------------------------
>>> x = 2
>>> y = 3
>>> z = -1
>>> x <= y <= z #instruction de tests multiples : chaque variable n'est
    évaluée qu'une seule fois
False
##------------------------------------------------------------------------
# Évaluation paresseuse de Python
>>> L = [1,2,3]
>>> (0>1) and (L[10]>0) #évaluation paresseuse de Python : (0>1) est
    faux, donc L[10] n'est pas évalué (alors que L[10] n'est pas défini)
    car le résultat est forcément "False"
False

>>> (2<3) or (L[10]>0) #évaluation paresseuse de Python : (2<3) est vrai
    , donc L[10] n'est pas évalué (alors que L[10] n'est pas défini) car
    le résultat est forcément "True"
True
```

Il est indispensable de connaître les tables de vérité de `or` et de `and` en ayant conscience de l'évaluation dite *paresseuse* de Python :

x	y	x or y
True	True*	True
True	False*	True
False	True	True
False	False	False

*non évalué en Python

x	y	x and y
True	True	True
True	False	False
False	True*	False
False	False*	False

*non évalué en Python

Il existe huit opérateurs de comparaisons :

x==y	x est égal à y (il y a 2 fois le signe égal)
x!=y	x différent de y
x>y	x strictement supérieur à y
x>=y	x supérieur à y
x<y	x strictement inférieur à y
x<=y	x inférieur à y
x is y	x et y sont des alias du même objet
x in y	x appartient à y

III Les entrées et les sorties

III.1 L'alimentation des données en entrée

L'alimentation des données en entrée diffère selon que :

- la donnée est saisie par l'utilisateur,
- la donnée est initialisée dans le code,
- il s'agit de l'ouverture d'un fichier ou d'une base de données.

Dans ce paragraphe, nous nous focalisons uniquement sur la saisie d'une donnée par l'utilisateur via l'instruction `input`.

```
1  nom = input("Entrez votre nom: ") # par défaut, input renvoie une chaîne de
       caractères
2
3  age = int(input("Entrez votre âge: ")) # int transforme la chaîne de caractè
       res en entier
4
5  taille = float(input("Entrez votre taille: ")) # idem avec float
6
7  x = eval(input("Entrez l'abscisse : ")) # eval évalue l'expression dans la
       chaîne de caractères saisie par l'utilisateur. Par exemple, on peut saisir
        1/3 ou sqrt(2)
8
9  x = eval('2*3') # alimente x avec l'entier 6
```

III.2 L'affichage des données en sortie

L'affichage à l'écran des données en sortie se fait via l'instruction **`print`**.
Les différents modes d'utilisation de **`print`** sont les suivants :

```
1  x = 21
2
3  print("le double de ",x," vaut ",2*x)
4  >>> (executing lines 1 to 2 of "<tmp 1>")
5  le double de  21  vaut  42  # notez le caractère blanc ajouté avant et après
       la variable
6
7  print("le double de "+str(x)+" vaut "+str(2*x))
8  >>> (executing lines 1 to 2 of "<tmp 1>")
9  le double de 21 vaut 42 # l'affichage est sans caractère additionnel
10
11 print("le double de %i vaut %i"%(x,2*x)) # le format est proche de celui du
       langage C
12 >>> (executing lines 1 to 2 of "<tmp 1>")
13 le double de 21 vaut 42
14
15 print("le double de {0} vaut {1} et non pas {0}, bien entendu !".format(x,2*x
       )) # le format est du pur Python; {0} fait référence à la variable indicée
        0 de la fonction "format", càd. x ici
16 >>> (executing lines 1 to 2 of "<tmp 1>")
17 le double de 21 vaut 42 et non pas 21, bien entendu !
```

Nous accepterons tous les modes d'utilisation. Par défaut, nous utiliserons le premier ou le dernier mode en Travaux Pratiques.

IV Les fonctions

IV.1 Généralités

Il est possible de définir (avec une syntaxe proche des mathématiques) une **fonction** qui, à partir d'un ou plusieurs arguments d'entrée renvoie un ou plusieurs arguments de sortie. Pour cela, on utilise le mot-clé **def** qui permet de définir une fonction, ainsi que le mot-clé **return** qui renvoie en sortie un objet (un nombre, un tableau, une chaîne de caractères,...).

Remarques :

- Le mot-clé **return** n'est pas une fonction ! D'où l'absence de parenthèses après le **return**.
- Il est possible de ne pas écrire de **return** en fin de fonction, typiquement lorsqu'on écrit une procédure. Mais en réalité, une fonction Python renvoie toujours une valeur. En l'absence de **return**, la fonction renvoit implicitement la valeur None.
- Ne confondez pas **print** (on affiche et on oublie) et **return** (on utilise le résultat envoyé par la fonction par la suite).

▶ *Exemple 1 :*

```
def cube(x):
    '''renvoie le cube du nombre x'''
    return x**3
```

On peut ensuite appeler la fonction dans le shell (i.e. la console) :

```
>>> cube(-1)
-1

>>> cube(2)
8

>>> help(cube)
Help on function cube in module __main__:
cube(x)
    renvoie le cube du nombre x
```

▶ *Exemple 2 :* Il est possible de définir une fonction avec des paramètres par défaut.

```
def puissance(x,n=0):
    '''renvoie x à la puissance n'''
    return x ** n
```

On peut ensuite appeler la fonction dans le shell (i.e. la console) :

```
>>> puissance(2) #La puissance par défaut est 0
1

>>> puissance(2,10) #2 à la puissance 10 (version 1)
1024

>>> puissance(2,n=10) #2 à la puissance 10 (version 2)
1024
```

▶ *Exemple 3 :* Une fonction peut appeler une ou plusieurs autres fonctions.

```
def mini2(x,y):
    '''renvoie le plus petit des deux nombres x et y'''
    if x < y:
        return x
    return y

```

```
def mini3(x,y,z):
    '''renvoie le plus petit des trois nombres x, y et z'''
    return mini2(mini2(x,y),z)
```

On peut ensuite appeler la fonction dans le shell (i.e. la console) :

```
>>> mini2(41,-2)
-2
>>> mini3(41,-2,12)
-2
```

IV.2 La « chaîne de documentation » et les commentaires

- La **chaîne de documentation** (ou "docstring" dans la langue de Shakespeare) d'une fonction est une chaîne de caractères encadrée par des triples guillemets (pour la distinguer des autres chaînes de caractères) : elle explique comment utiliser la fonction, les arguments d'entrée attendus et les valeurs retournées (*postconditions*) ainsi que les conditions d'utilisation de la fonction (*préconditions*).
  La docstring d'une fonction est accessible via la commande *help*. Toutes les fonctions standards des modules de Python contiennent une docstring.

  ▶ *Exemple :*

```
def indice_maximum_tableau(t):
    '''renvoie l'indice de la valeur maximale du tableau.
        Préconditions : le tableau t est supposé non vide.
        Postconditions : le nombre renvoyé est un entier compris
   entre 0 et len(t) exclu'''
    m = 0
    for i in range(len(t)):
        if t[i] > t[m]:
            m = i
    return m

#dans le shell :
>>> help(indice_maximum_tableau)
Help on function indice_maximum_tableau in module __main__:

indice_maximum_tableau(t)
    'renvoie l''indice de la valeur maximale du tableau. Pré
   conditions : le tableau t est supposé non vide. Postconditions :
   le nombre renvoyé est un entier compris entre 0 et len(t) exclu'
```

- Les **commentaires** du code expliquent comment fonctionne le code. Les commentaires ne sont pas lus par l'interpréteur du code. Ils sont destinés à faciliter la compréhension du code et sa maintenance. Ils démarrent par le symbole `#`.
  Cette année, les programmes seront commentés avec des éléments (invariants, variants,...) prouvant la validité de certaines parties du code.
  On commentera ce qui ne se déduit pas automatiquement par la lecture du code, mais on veillera à ne pas "surcommenter".

  ▶ *Exemple :*

```
n = 25 #le nombre de nombres premiers inférieurs à 100 est bienvenu
x = x + 1 #incrémenter x est un commentaire inutile
```

On se rappellera de la citation suivante :
"Les programmes doivent être écrits pour être lus par des gens et accidentellement exécutés par des machines." Abelson & Sussmann, *Structure and Interpretation of Computer Programs.*

V Les tests conditionnels

Un test conditionnel évalue une expression booléenne et réalise ou non une suite d'instructions (on dit **un bloc** d'instructions) selon la valeur de vérité de l'expression booléenne.
La structure d'un test conditionnel varie selon la présence ou non d'alternatives. On peut distinguer trois situations :

1er cas :

```
if booléen:
    instructions
```

Le simple fait d'avoir écrit ":" juste après le booléen, a permis d'**indenter** les instructions du bloc `if`. Si le booléen vaut `True`, les instructions indentées sont exécutées.

2e cas :

```
if booléen:
    instructions
else:
    instructions alternatives
```

Si le booléen vaut `True` alors on réalise **uniquement** le bloc d'instruction du `if`, sinon on réalise le bloc d'instructions alternatives.

3e cas :

```
if booléen:
    instructions
elif booléen 1: #exécuté quand booléen == False et booléen 1 == True
    instructions alternatives 1
elif booléen 2:
    instructions alternatives 1
...
else:
    instructions alternatives n
```

Le mot clé `elif` est la contraction de `else if`.
Les booléens sont évalués dans l'ordre de lecture. Dès que l'un deux prend la valeur `True`, le bloc d'instructions correspondant est exécuté. Dans le cas où aucun booléen ne vaut `True`, le bloc d'instructions du `else` est exécuté.

▶ *Exemple :*

```
a = 1
b = 3
if a > 0:
    a = a + 1
elif b <= 4:
    b = b - 1
```

A la fin de l'exécution de ce programme, comprenez bien que la variable a vaut 2 et la variable b vaut 3. En effet, le bloc d'instructions associé au `elif` n'est jamais exécuté puisque a n'est pas négatif. Ainsi la variable b n'est pas modifiée !

VI Les structures de boucle

VI.1 Les boucles bornées

Les **boucles bornées** permettent de répéter un bloc d'instructions. On peut distinguer trois modes d'utilisation de la boucle bornée :

1er cas : la boucle simple

```
n=...#n correspond au nombre de fois que l'on souhaite réaliser le bloc d'
    instructions
for _ in range(n): # on n'utilise pas le compteur de boucles, donc on écrit _
    instructions
```

▶ *Exemple : Calcul d'une somme de données saisies par l'utilisateur*

```
n = int(input("Entrer le nombre d'élèves: "))
somme = 0

for _ in range(n):
    k = int(input("Entrer une valeur: "))
    somme = somme + k

print("La somme est: ", somme)
```

▶ *Exemple : Initialisation d'une ligne de 0*

```
n  = int(input())
ligne = []
for _ in range(n):
    ligne.append(0)
```

2e cas : la boucle avec un indice

```
#n est initialisé au nombre de fois que l'on souhaite réaliser le bloc d'
    instructions avant la boucle for
for i in range(n): #i est une variable muette, tout autre nom autorisé
    convient
    instructions
```

Il s'agit typiquement des situations où l'on souhaite :

- faire la somme ou le produit de termes d'une suite
- parcourir un tableau ou une chaîne de caractères

L'objet `range(n)` se comporte comme une fonction ; il ne "contient" aucun nombre. Utilisé par `for`, il génère successivement tous les entiers de **0 inclus** à $\boldsymbol{n}$ **exclu**, i.e. de **0 à** $\boldsymbol{n-1}$.

Plus précisément, soit $a < b$:

- L'objet `range(a,b)` génère successivement tous les entiers de $\boldsymbol{a}$ **inclus** à $\boldsymbol{b}$ **exclu**, i.e. de $\boldsymbol{a}$ **à** $\boldsymbol{b-1}$.
- L'objet `range(a,b,pas)` génère successivement tous les entiers de $\boldsymbol{a}$, $\boldsymbol{a+pas}$, $\boldsymbol{a+2*pas}$ jusqu'à $\boldsymbol{b}$ **exclu**, i.e. de $\boldsymbol{a}$ **à** $\boldsymbol{k*pas}$ avec k le plus grand entier tel que $a + k * pas < b$.

Remarque : quand $a \geq b$, il n'y a pas de message d'erreur : la boucle n'est pas exécutée.
Si l'on souhaite obtenir des indices décroissants, on choisit $a > b$ **et** on indique un pas négatif.

Si nécessaire (voir 3e cas ci-dessous), l'objet `range(n)` peut être remplacé par une *séquence* telle qu'un tableau, un dictionnaire, un ensemble, un tuple (généralisation de la notion de couple) [3].

3. `range(n)` ainsi que les séquences mentionnées ci-dessus sont appelées des "itérables" ; `for` produit un itérateur dont la seule fonction est d'appeler succesivement les éléments de l'itérable considéré.

▶ *Exemple : Recherche d'un élément dans un tableau*

```
def recherche(t,x):
    '''renvoie True si le nombre x est dans le tableau t et False s'il ne s'y trouve pas. Le tableau t peut être vide.'''
    n = len(t)
    for i in range(n):   #on parcourt tous les éléments du tableau (par indice)...
        if t[i] == x:    #...et on teste si chaque élément est égal à x
            return True  #si tel est le cas, on renvoie True
    return False         #si aucun d'entre eux n'est égal à x (ce qui est le cas en fin de boucle), on renvoie False
```

3e cas : la boucle avec un élément

Lorsque l'indice d'un tableau semble artificiel pour la fonction à réaliser, on peut penser à ce type de boucles.

▶ *Exemple : Somme des éléments d'un ensemble*

```
def somme_tableau(e): #exemple, e={1,2,3}
    '''renvoie la somme des valeurs numériques de l'ensemble e. Si celui-ci est vide, renvoie 0 par défaut.'''
    somme = 0 #la variable somme est appelée "un accumulateur"
    for nb in e: #quand l'ensemble n'est pas vide, nb est une valeur de l'ensemble. Quand l'ensemble est vide, la boucle n'est pas exécutée.
        somme = somme + nb #ou aussi : somme += nb
    return somme
```

Il existe une fonction `sum` qui réalise automatiquement cette somme pour les objets de type `list`. Cependant, sauf si mentionné explicitement dans un énoncé, il sera interdit de l'utiliser en concours.

VI.2 Les boucles conditionnelles

Lorsque l'on connait précisément le nombre de boucles à exécuter (éventuellement dépendant d'un paramètre n), on utilisera systématiquement la boucle bornée **`for`**. Cependant, il est parfois utile de vouloir exécuter un traitement tant qu'une condition est vérifiée. Le traitement s'arrête donc quand la condition n'est plus vérifiée. Pour ce faire, on utilise une boucle conditionnelle comme suit :

```
initialisation de variables
while booleen : #ce boléeen utilise l'une des variables initialisée
    instructions #l'une des instructions doit mettre à jour une variable utilisée par le booléen pour arrêter le traitement au moment voulu
```

- Lorsque la valeur du booléen n'est jamais modifiée par le bloc d'instructions qui suit la boucle **`while`**, on parle de **boucle infinie** : le code ne permet pas de sortir de la boucle qui se répète indéfiniment (l'écran du shell reste "blanc").
- Le booléen d'une boucle **`while`** doit toujours être initialisé à `True` : c'est la première chose à vérifier.

▶ *Exemple d'une suite numérique : recherche d'un seuil*

```
#Etude de la suite (u(n)) définie sur N par u(n+1) = 0.9 * u(n) + 3 et u(0) = 40.
#Il s'agit d'une suite décroissante convergeant vers 30.
#On cherche la plus petite valeur de n pour laquelle u(n) < 32.

n = 0
u = 40 #il s'agit de u(0) car n == 0
```

```
while u >= 32:
    n = n + 1
    u = 0.9 * u + 3 #la variable u de gauche correspond à u(n) avec n>=1
#Ici, après la boucle, on a (u<32): on décide donc d'afficher n.
print(n)
```

Notez que, dans le programme précédent, le booléen $(u >= 32)$ vaut `True` après l'initialisation de u par 40. Par ailleurs, la valeur de u étant modifiée par l'instruction $u = 0.9 * u + 3$, le booléen $(u >= 32)$ est modifié à chaque boucle. Cependant, dans le cas présent, seule une étude mathématique permet de savoir qu'il existe une valeur de u pour laquelle le booléen $(u >= 32)$ sera `False`. En général, **le problème de la terminaison d'une boucle** devra être étudié avec soin.

Références

[1] Thibaut Balonski, Sylvain Conchon, Jean-Christophe Filliâtre, Kim Nguyen *Numérique et Sciences Informatiques - Spécialité NSI 1re et Tle.*

[2] Kamal Haïdar *Informatique pour Tous - Cours et exercices.*